Wiskunde en Informatica

Bart Demoen
KU Leuven
Departement Computer Wetenschappen

De activiteit van de informaticus is sterk gericht op het maken van *software*. Daartoe behoort o.a. het uitschrijven van een algoritme in een programmeertaal. In deze bijdrage ligt de focus op algoritmen, en we gebruiken een bijna-echte programmeertaal om die te beschrijven. Dat levert *pseudo-code* op, en die geeft ons een heel goede context om met behulp van wiskunde aan te tonen dat die algoritmen *werken*: we hebben het over correctheid, eindigheid en efficiëntie van algoritmen.

Deze bijdrage is als volgt opgebouwd: we beginnen met duidelijk te stellen welke taal hier gebruikt wordt om algoritmen te beschrijven. Wie al geprogrammeerd heeft in C, Pascal, Java ... herkent zeker de concepten en zelfs wat syntax. Daarna komt het algoritme van Euclides aan bod: het berekent de grootste gemene deler van twee getallen. De eindigheid en de correctheid ervan worden uitgewerkt. Daarbij steunen we op rekenkunde, een stukje wiskunde waarmee leerlingen in het secundair vertrouwd zijn. Het algoritme van Prim dat daarna behandeld wordt, steunt op grafentheorie. Dat stukje wiskunde is ook bij leerkrachten wiskunde dikwijls minder bekend, maar het is heel toegankelijk, en levert bovendien algoritmen op die dagelijks van belang zijn. In de sectie erna tonen we een wiskundige manier om de efficiëntie van algoritmen te bestuderen en af te leiden. Daarbij gebruiken we recurrentievergelijkingen, een discrete versie van differentiaalvergelijkingen. Daarna gaan we kort in op eindigheid van algoritmen.

Op het einde geven we een kort overzicht van andere toepassingen van wiskunde in de informatica en gaat het even over de toepassing van informatica in de wiskunde.

Voor we eraan beginnen nog dit: een informaticus wordt wel eens een *mathematician in a hurry* genoemd. Daar is wel iets van: wij moeten de formele aspecten van onze activiteit in het oog houden - en daarbij komt wiskunde van pas - maar wiskunde is niet ons doel of onze hoofdinteresse en daarom lopen we er al eens de kantjes af. Goed dat er altijd echte wiskundigen bij de hand zijn om ons op het rechte pad te houden.

Algoritmen beschrijven

Een algoritme is iets abstracts: een procedure die stap voor stap voorschrijft wat moet gedaan worden, en die eindigt. De procedure voor de staartdeling is een mooi voorbeeld van een algoritme. Een programma is meer concreet: het is geschreven in een programmeertaal en hoeft niet te eindigen. Een programma kan een algoritme implementeren. Dat

onderscheid tussen algoritme en programma is hier niet zo belangrijk. We tonen verschillende algoritmen, en we beschrijven die in een pseudo-programmeertaal. Daarmee kunnen we de essentie van een algoritme laten zien zonder alle spitsvondigheden van een echte programmeertaal te moeten uitleggen. We spreken wel goed af wat de symbolen en constructies in onze pseudo-taal betekenen en welke de concepten zijn.

Het eerste begrip is dat van *programmavariabele* en *toekenning*: we gebruiken letters (of rijen van letters) om een variabele aan te duiden. Zo is bijvoorbeeld a een variabele met naam 'a'. Om die variabele een waarde (bijvoorbeeld 7) te geven, schrijven we de toekenning $a = 7$. We lezen dat als *a krijgt de waarde 7*. Die a kan je zien als de naam van een doos, waarin na de toekenning de waarde 7 zit. We kunnen in die doos later een nieuwe waarde steken, door een nieuwe toekenning te doen: de opeenvolging van toekenningen

$a = 7; a = 9; a = -1;$ heeft als uiteindelijk resultaat dat de a-doos de waarde -1 bevat.

Een programma gebruikt meestal meer dan één variabele. Hieronder een paar programmalijnen die a, b en c gebruiken:

```
a = 7;
b = 9;
c = 12;
a = c - b;
```

De eerste drie lijnen *initialiseren* a, b en c. De laatste lijn vervangt de waarde die in a zat door het verschil van de waarden die in c en b zitten. Op het einde van de uitvoering van dit programma zit in a de waarde 3, in b 9 en in c 12. We zeggen gewoon dat a gelijk is aan 3, enzovoort.

Merk op dat er een verschil is tussen de toekenning = en de wiskundige gelijkheid =: in de wiskunde is $a = a + 1$ niet waar, en in de informatica vervangt de toekenning $a = a + 1$ de waarde die in de a-doos zat door één meer.

Behalve toekenningen en rekenkundige bewerkingen gebruikt een programma ook het *testen op gelijkheid*: zo kunnen we testen of a gelijk is aan 19 door te schrijven $a == 19$. Dat is waar of onwaar en we gebruiken zulk een test in de *if-then-else* constructie. Hier is een voorbeeld:

```
<een initialisatie van a>
if (a == 19)
    then b = 10;
    else b = 20;
```

Afhankelijk van de initialisatie van a is b op het einde van het programma gelijk aan 10 of 20: als de test $a == 19$ waar is (slaagt) dan wordt de code in de then-tak uitgevoerd, anders de code in de else-tak. Als in een tak meer dan één toekenning (of instructie) uitgevoerd moet worden, dan gebruiken we accolades { } om dat duidelijk te maken, bijvoorbeeld:

```
if (a > 19)
    then { b = 10; a = 0; }
    else { b = 2*a ; a = 1; }
```

In het voorbeeld zie je ook dat we niet alleen gelijkheid kunnen testen, maar ook strikt-groter-dan, en de andere varianten uit de rekenkunde kunnen ook. We gebruiken <> voor de ongelijkheid.

We hebben nog een constructie nodig: de *herhaling-zolang-een-voorwaarde-voldaan-is*. We noemen dat een *lus*. Een voorbeeld moet dat duidelijk maken. Wat achter een % staat in pseudo-code is commentaar: het beïnvloedt het

programma niet, maar geeft uitleg, meestal over de programmalijn waarop die commentaar staat.

```
       a = 5; b = 0;     % initialisatie van a en b
       while (a > 0) do
          { b = b + a;
            a = a - 1;
          }
```

De lus in woorden: zolang a strikt groter is dan nul, tel de (huidige) waarde van a op bij b en verminder a met één. Met de gegeven initialisatie moet je nu kunnen inzien dat op het einde van het programma de volgende twee gelijkheden waar zijn:

```
       a == 0
en     b == 15
```

Dat laatste is waar want 0+5+4+3+2+1 == 15. Een programma berekent meestal iets uit een aantal gegeven grootheden. Het moet duidelijk zijn welke die gegevens zijn. We tonen dat door een stukje programma een naam te geven, alsof het een functie is, en de gegevens mee te geven aan die functie. Bovendien moet de *uitkomst* op een of andere manier ook veruitwendigd worden. Dat doen we m.b.v. de *return* instructie. Zo kunnen we het laatste voorbeeld herschrijven als

```
    stukjeprogramma(a):  % a is gegeven
        b = 0;           % initialisatie van b
        while (a > 0) do
           { b = b + a;
             a = a - 1;
           }
        return b;        % b wordt teruggegeven
```

Bij de uitvoering van de instructie *return* stopt de procedure, en de waarde in het argument van *return* wordt teruggegeven aan wie het stukje programma opriep. Van dit programma met naam *stukjeprogramma* kunnen we zeggen: gegeven de waarde 5, schrijft het de waarde 15 uit. Of meer algemeen: als de gegeven waarde N is, dan is de uitgeschreven waarde $N(N+1)/2$. Bovenstaande notatie beschouwt dus *stukjeprogramma* als een functie met één argument, en wat teruggegeven wordt is de functiewaarde.
We zijn nu klaar voor het eerste algoritme.

Rekenkunde voor het algoritme van Euclides

Twee waarden (natuurlijke getallen) zijn gegeven, en hun grootste gemene deler (GGD) moet berekend worden. Één versie van het algoritme van Euclides is gebaseerd op modulo rekenen. We tonen pseudo-code voor een andere versie, gebaseerd op herhaald aftrekken:

```
    ggd(a,b) :
        while (a <> b) do
              if (a > b)
                    then a = a - b;
                    else b = b - a;
        return a;           % of b, maakt niet uit
                            % ze zijn toch gelijk
```

Let op: met $GGD(a,b)$ bedoelen we de grootste gemene deler van a en b. Met $ggd(a,b)$ bedoelen we de waarde die de bovenstaande functie *ggd* teruggeeft: we moeten nog bewijzen dat $ggd(a,b) == GGD(a,b)$.
Euclides beschreef dit algoritme rond 300 v. Chr.. Twee voorbeelden volgen hoe de waarden van a en b veranderen tijdens de uitvoering van het algoritme:

```
                     a     b   |   a      b
initieel            21    14   |  36     48
na 1 lusdoorgang     7    14   |  36     12
na 2 lusdoorgangen   7     7   |  24     12
na 3 lusdoorgangen               |  12     12
```

Het eindigt telkens met $a == b$ en die waarde is bovendien de GGD van de gegeven getallen.
Meestal wordt het juist werken van een algoritme opgesplitst in twee eigenschappen die dan apart formeel worden bewezen:

- eindigheid: het algoritme eindigt (na een eindig aantal stappen) voor elke geldige input
- correctheid: als het algoritme eindigt (t.t.z. het stopt na een eindig aantal stappen) dan produceert het een correct resultaat

Er is een goede reden om die twee eigenschappen uit elkaar te trekken: de technieken om ze te bewijzen verschillen.
Om de eindigheid te bewijzen voegen we wat formele commentaar toe aan het programma: die commentaar geeft aan wat we weten over de variabelen in het programma, eventueel in relatie tot de grootheid die we willen berekenen. We schrijven die formele commentaar tussen [ ], en we spreken over *programma-invarianten*: die commentaren, die uitspraken, zijn invariant t.o.v. de uitvoering van het programma. Of om het nog anders te zeggen: elke keer als de uitvoering van het programma die uitspraak passeert, is die uitspraak waar.

```
ggd(a,b):
  [a > 0; b > 0; s = a + b; s > 0]
  while (a <> b) do
    [a <> b, dus a > b ofwel b > a]
    if (a > b)
     then a = a - b; [a > 0; b > 0; s = a + b; s > 0;
                      s strikt kleiner dan voorheen]
     else b = b - a; [a > 0; b > 0; s = a + b; s > 0;
                      s strikt kleiner dan voorheen]
  [a > 0; b > 0; a == b]
  return a;
```

s is een hulpveranderlijke die we gebruiken in het bewijs: we zorgen dat s steeds gelijk is aan $a+b$, dus telkens als a of b verandert, passen we s weer aan. Dat de invarianten waar zijn, kan je zeker inzien. Voor de eindigheid redeneren we nu als volgt: tijdens elke uitvoering van de lus wordt s strikt kleiner, maar zeker niet nul. Vermits in $\mathbb{N}$ geen oneindige strikt-dalende rij bestaat eindigt het programma. Die redenering is meteen de meest gangbare voor het bewijzen van de eindigheid van een stukje programma: er is een strikt dalende grootheid in een verzameling met een orde die geen oneindig dalende rijen toelaat, dus eindigt het programma. Dit laat bovendien zien waarom eindigheid niet gegarandeerd is voor *slechte* input: inderdaad, als je $ggd(\sqrt{2}, 1)$ uitvoert, dan werkt het stopargument niet, want in $\mathbb{R}_+$ bestaan er wel oneindige strikt dalende rijen. Je kan bovendien nagaan dat $ggd(\sqrt{2}, 1)$ niet stopt, omdat $\sqrt{2}$ en 1 onderling onmeetbaar zijn.
Nu bewijzen we de correctheid. We voegen opnieuw wat invarianten toe aan het programma, en redeneren daarmee:

```
ggd(a,b):
   [stel g gelijk aan de grootste gemene deler
    van de gegeven a en b: g = GGD(a,b)]
   while (a <> b) do
         if (a > b)
               then a = a - b; [g == GGD(a,b) *]
               else b = b - a; [g == GGD(a,b) **]
   [a == b, dus GGD(a,b) == a;
    bovendien: g == GGD(a,b), dus g == a]
   return a;
```

g is geen programma-variabele, maar een grootheid die we gebruiken voor het bewijs. g verandert niet tijdens het programma (of het bewijs): het is de waarde van de grootste gemene deler van de gegeven a en b. We willen bewijzen dat het deze waarde is die teruggegeven wordt door het programma.
De sleutel van het bewijs zit in de uitspraken aangeduid met * en **: waarom is na de

toekenning $a = a-b$ (als $a > b$) g nog altijd gelijk aan $GGD(a, b)$? De reden is de wiskundige
Stelling
Als $a > b$ dan $GGD(a, b) == GGD(a - b, b)$
Andere stukjes van het bewijs hebben we in het programma tussen [ ] gezet.

Wat hebben we gezien? Van een heel oud algoritme hebben we de correctheid en de eindigheid bewezen: we steunden daarbij op eigenschappen van natuurlijke getallen, van de grootste gemene deler, en op redeneringen die in de wiskunde aanvaard zijn bij het bewijzen van stellingen. Daarbij moet de toekenning toch wat nader bekeken worden: de toekenning speelt een rol in het dynamisch proces dat door een programma beschreven wordt, terwijl een stelling in de wiskunde klassiek meestal een statische waarheid beschrijft. Dat in overeenstemming brengen heeft wiskundigen/informatici heel wat tijd gekost.
Als afsluiter van deze sectie, een nieuwe versie van het ggd-algoritme:

```
ggd(a,b):
  if (a == b) then return a;
  else if (a > b) then return ggd(a-b,b);
  else return ggd(a,b-a);
```

Deze versie heet *recursief* omdat de functie ggd zichzelf oproept. Probeer de opeenvolging van oproepen van ggd te bepalen voor ggd(21,14). Je moet krijgen:

```
ggd(21,14)
ggd(7,14)
ggd(7,7)
```

Welke waarden geven die oproepen terug?

Wat komt er nog? Hierboven werd een algoritme beschreven voor een probleem uit de rekenkunde: het berekenen van een grootste gemene deler. Dat daarbij kennis (een stelling) uit de rekenkunde belangrijk was, ligt nogal voor de hand. Wat nu volgt is een probleem uit een heel ander domein, een algoritmische oplossing, een bewijs van correctheid/eindigheid, een reden waarom het efficiënt is, en nog een paar andere verrassingen ...

Grafentheorie voor correcte, efficiënte algoritmen

Het probleem Tussen een aantal steden lopen tweerichtingswegen met een middenberm. Je kan bovendien van elke stad naar elke andere stad rijden langs die wegen. We willen *voldoende* middenbermverlichting voorzien op die wegen en aan de laagst mogelijke *kost*. Voldoende betekent hier dat het mogelijk is om van elke stad naar elke andere stad te rijden op verlichte wegen. De kost van het verlichten van een weg is evenredig met de lengte van die weg.
We zoeken dus een deel van het gegeven wegennetwerk dat alles nog verbindt en dat minimale totale lengte heeft.
We zetten eerst het bovenstaande probleem over naar een abstracte context: die van grafentheorie. Een graaf G is zoals een wegennetwerk: er zijn knopen (de steden) en bogen (de wegen). De verzameling knopen noteren we door V. Een boog tussen twee knopen a en b noteren we door de verzameling $\{a, b\}$: die knopen zijn de uiteinden van de boog. De verzameling bogen noteren we door E en dus is $E \subseteq \{\{a, b\} | a, b \in V\}$. Omdat we ook willen modelleren dat de wegen een lengte hebben, gebruiken we bovendien een gewichtsfunctie w :

$E \to \mathbb{N}$. Een graaf G is dan het drietal (V, E, w). Het gewicht van een graaf (V, E, w) is de totale weglengte in de graaf en gedefiniëerd als $w(G) = \sum_{e \in E} w(e)$.

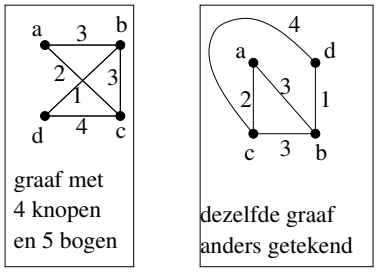

Figure 1: Een gewogen graaf

Figuur 1 toont twee verschillende tekeningen van dezelfde graaf: het is inderdaad niet belangrijk hoe een graaf getekend wordt, en verschillende tekeningen van dezelfde graaf kunnen onze intuïtie over verschillende eigenschappen ondersteunen.

Enige begrippen uit grafentheorie Een graaf $G' = (V', E', w)$ is een *deelgraaf* van graaf $G = (V, E, w)$ indien $V' \subseteq V$ en $E' \subseteq E$. Een deelgraaf $G' = (V', E', w)$ van graaf $G = (V, E, w)$ *spant* G *op* indien $V' = V$.
Figuur 2 toont een graaf, een opspannende en een niet-opspannende deelgraaf ervan.
In een graaf $G = (V, E, w)$ is een *pad* tussen knopen a en b een opeenvolging van knopen $x_1, x_2, ..., x_n$ met $n > 1$ zodanig dat $x_1 = a$, $x_n = b$ en $\{x_i, x_{i+1}\} \in E$ voor $i = 1..(n-1)$. Een pad is een *kring* als $x_1 = x_n$. Een graaf heet *volledig verbonden* of *samenhangend* als er tussen elke twee knopen een pad bestaat. Een *boom* is een samenhangende graaf zonder kring.
Figuur 3 toont drie bomen, maar zonder de namen van de knopen of de gewichten.

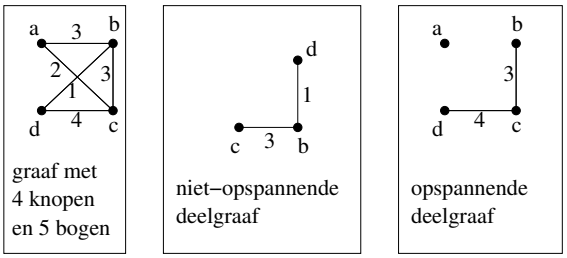

Figure 2: Gewogen graaf en deelgrafen

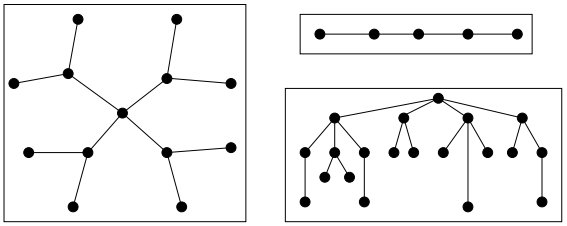

Figure 3: Drie bomen

Omdat bomen zo belangrijk zijn voor het vervolg, geven we nog - zonder bewijs - drie equivalente definities van een boom: een graaf met n knopen is een boom indien (1) de graaf $n-1$ bogen heeft en samenhangend is, of (2) de graaf $n-1$ bogen heeft en kringloos, of (3) tussen elke twee knopen een uniek pad bestaat.
Als je een boog wegneemt uit een boom, dan is die niet meer samenhangend. Als je een boog toevoegt, dan is de graaf niet meer kringloos. Probeer dat uit op de getekende bomen.

Het probleem in termen van grafen De overeenkomst tussen een wegennetwerk en een gewogen graaf is duidelijk. De graaf maakt abstractie van de concrete vorm van het wegennetwerk. Het vinden van een beste oplossing voor het probleem van de middenbermverlichting is nu: gegeven een graaf

$G = (V, E, w)$, vind een deelgraaf T van G zodat T een boom is, T G opspant en T minimaal gewicht heeft van alle bomen die G opspannen. Zulk een deelgraaf noemt men een minimaal opspannende boom (van G), afgekort een MOB. Waarom werken we liever met de abstractie (de graaf) dan met het wegennetwerk? Één reden is alvast dat we gelijkaardige problemen nu gemakkelijk herkennen als hetzelfde probleem: het kon net zo goed over de aanleg van een elektriciteitsnetwerk gaan in een stad, van de waterleidingsbuizen in een huis, of over de studie van een chip circuit, of van een Facebook netwerk.

Een programma dat het probleem oplost Een correct en eindig programma is gemakkelijk geschreven: we beschrijven het met minder details dan gewoonlijk, bijna in woorden. Daarbij is de gegeven G een samenhangende graaf.

```
mob(G):
        construeer alle opspannende bomen T van G;
        bereken voor elk ervan het gewicht w(T);
        return een met kleinste gewicht;
```

Het systematisch genereren van alle opspannende bomen van G is niet triviaal. Bovendien willen we het eigenlijk niet doen, want ... een graaf G met n knopen heeft mogelijk tot n^{n-2} verschillende opspannende bomen: dit werd bewezen door de wiskundige A. Cayley (1821-1895). Het bovenstaande programma kan dus n^{n-2} operaties nodig hebben (of meer). België heeft 134 steden ... tel maar na. Het aantal operaties in een algoritme bepaalt de tijd die een computerprogramma nodig heeft voor de uitvoering ervan, en is dus een goede maat voor de efficiëntie van het algoritme. De bovenstaande aanpak om een MOB te berekenen is dus heel inefficiënt. Maar het heeft ons toch iets opgeleverd: er bestaat altijd een MOB. Probeer dat in te zien.
De eerste die het MOB probleem algoritmisch efficiënt oploste was O. Borůvka: hij deed dat in 1926 om een elektriciteitsnetwerk te maken voor Moravia. Hier tonen we het algoritme van Prim: het is eenvoudiger en geeft ons de gelegenheid om een duidelijk verband te tonen tussen een eigenschap van een MOB, die we eerst bewijzen, en de correctheid en de efficiëntie van het algoritme.

Eigenschap: uitbreiding van een partiële MOB Gegeven een samenhangende gewogen graaf $G = (V, E, w)$, en een deelboom T van G. Stel $e = \{a, b\}$ een boog van G met $a \in T$ en $b \notin T$ met kleinste gewicht van al zulke bogen. Dan geldt: indien T een deel is van een MOB van G, dan is ook $T \cup e$ deel van een MOB van G.

Intuïtie achter het bewijs Een MOB heeft zeker een boog y tussen T en daarbuiten. Het bewijs laat zien dat je die y kan vervangen door e en daardor opnieuw een MOB verkrijgt: nu zit e er zeker in.

Bewijs Noem de gegeven MOB waartoe T behoort M. Indien e al in M zit, dan hoeven we niks te bewijzen. Stel dus dat e niet in M zit. Dan hebben we een situatie zoals in figuur 4: de volle lijnen stellen de bogen van M voor. T bestaat uit de bogen in de linker ovaal. In de rechterovaal zitten de knopen van G die niet in T zitten. Er kunnen meerdere bogen (zoals x en y) zijn met een knoop in T en één erbuiten, en verschillend van e (er moet minstens één zo'n boog zijn - zorg dat je dat inziet).

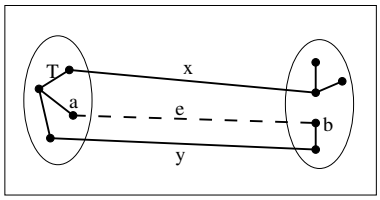

Figure 4: e behoort niet tot M

Voeg nu aan M de boog e toe, en noteer de resulterende graaf $M \cup e$: figuur 5 toont die.

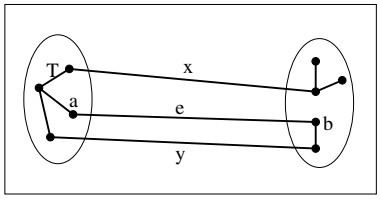

Figure 5: $M \cup e$

In M bestaat een pad van a naar b (want M is een opspannende boom), en door e toe te voegen komt er een nieuw pad bij, namelijk a, b. Dus $M \cup e$ bevat een kring waarin de boog e zit en ook een andere boog van T naar buiten T: in het geval van de tekening is dat y.
We verbreken die kring nu door uit $M \cup e$ die y te verwijderen (maar niet de knopen van y). We noteren de resulterende graaf door $(M \cup e) \setminus y$ en je verkrijgt de situatie van figuur 6.
Van $(M \cup e) \setminus y$ is het nu gemakkelijk in te zien dat het een boom is (want kringloos en samenhangend), dat hij G opspant, en bovendien is
$w((M \cup e) \setminus y) = w(M) + w(e) - w(y) \leq w(M)$
De laatste ongelijkheid is correct omdat e een kortste boog is van T naar erbuiten: dat was gegeven in de formulering van de eigenschap.
Dus $(M \cup e) \setminus y$ is een minimale opspannende

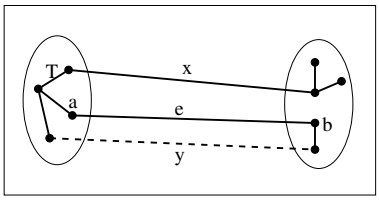

Figure 6: $(M \cup e) \setminus y$

boom van G waartoe e behoort. ■

De net bewezen eigenschap beschrijft een statisch gegeven: dat is typisch voor wiskundige stellingen. Het algoritme van Prim gebruikt die eigenschap op een dynamische manier om een MOB voor G te construeren. In woorden gaat dat als volgt: we beginnen met een triviale T, namelijk één knoop. T is deel van een MOB van G. Er bestaat een kortste boog van T naar een knoop niet in T, en die voegen we toe aan T: dankzij de bewezen eigenschap is T nog altijd een deel van een MOB van G. Dat proces herhalen we: voeg een kortste boog tussen T en daarbuiten toe aan T en blijf dat doen totdat T alle knopen bevat. Wat voortdurend waar is tijdens het algoritme is de uitspraak T *is deel van een MOB van* G. Op het einde van het algoritme bevat T alle knopen van G, dus is T opspannend, een boom, deel van een MOB, en dus zelf een MOB.

We beschrijven nu eerst het algoritme in een taal die iets minder formeel en van een hoger niveau is dan wat we voorheen gebruikten. Daarna komt een bewijs van correctheid. In het algoritme gebruiken we $knopen(H)$ waarbij H een graaf is: het is de verzameling van knopen in H.

```
prim(G): % G een gewogen samenhangende graaf (V,E,w)
    initialiseer T op ({v},{},w) waarbij
                v een willekeurige knoop in V is

    while (knopen(T) <> V) do
        bepaal een kortste boog e = {a,b} in E zo dat
             a in knopen(T) en b niet in knopen(T)
        % voeg e toe aan T
        T = T unie e;
    return T;
```

Figuur 7 laat voor een graaf zien hoe Prim een MOB construeert: de initiële graaf is getekend met volle lijnen voor de bogen, en volle cirkels voor de knopen. De gewichten staan bij de bogen. Telkens Prim een knoop of boog toevoegt aan de groeiende MOB wordt de knoop hol, en de boog een stippellijn. De eerste gekozen knoop is links boven in de getekende graaf. Er is dan direct al keuze tussen de twee bogen met gewicht 1. Probeer eens met een andere knoop te beginnen of andere keuzes te maken voor de bogen.

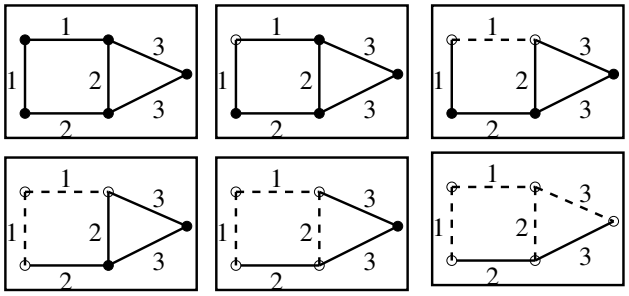

Figure 7: Prim in actie

De eindigheid van het algoritme bewijzen gaat als volgt: telkens de lus doorlopen wordt, komt er een knoop uit G bij in T, dus wordt de voorwaarde van de *while* ooit onwaar en stopt de *while* ooit. Eigenlijk kan je inzien dat de lus exact $\#V - 1$ keer uitgevoerd wordt.

De correctheid - namelijk dat op het einde T een MOB is - doen we door één of meer uitspraken op te stellen en te bewijzen dat die waar zijn op bepaalde programmapunten. Net zoals met het algoritme van Euclides voegen we die uitspraken toe aan het programma.

```
prim(G): % G een gewogen samenhangende graaf (V,E,w)
   initialiseer T op ({v},{},w) waarbij
               v een willekeurige knoop in V is
   [T is deel van een MOB van G]

   while (knopen(T) <> V) do
       [T is deel van een MOB van G *]
       bepaal een kortste boog e = {a,b} in E zodat
            a in knopen(T) en b niet in knopen(T)
       % voeg e toe aan T
       T = T unie e;
       [T is deel van een MOB van G **]

   [T is deel van een MOB van G *** en bevat heel V
       dus T is een MOB van G]
   return T;
```

Het belangrijkste stuk van het correctheidsbewijs is de overgang van * naar **: door e toe te voegen aan T zou de invariant geschonden kunnen worden, maar de bewezen eigenschap zegt dat die waar blijft.
In *** is alles waar wat op het einde van de uitvoering van de code binnen de *while* waar was, en bovendien ook dat $knopen(T) == V$: het besluit is dus dat Prim een MOB aflevert.

De complexiteit van Prim Vermits we vroeger schreven dat een naieve implementatie (probeer alle opspannende bomen en onthoud de kleinste) niet efficiënt genoeg is, voelen we ons verplicht om hier toch iets te zeggen over de complexiteit van het algoritme van Prim. Het is niet gemakkelijk om een optimale bovengrens voor het aantal stappen van het algoritme te geven, maar een ruwe bovengrens kan wel. Zoals voorheen gebruiken we n voor het aantal knopen

in de graaf. De *while* lus wordt dan $(n-1)$ keer uitgevoerd. Binnen in de lus zoeken we een kortste boog in E met de eigenschap dat ie T verbindt met erbuiten. Vermits er niet meer dan $n*(n-1)/2$ bogen kunnen zijn in een graaf met n knopen, kost dat zoeken van een kortste boog minder dan n^2 stappen, waarbij elke stap een constant aantal operaties vraagt (zie *Sequentieel Zoeken* elders in deze Loep). In combinatie met het toevoegen van een element aan een verzameling (bv. w aan $knopen(T)$), krijgen we een veilige bovengrens van n^4 operaties op een constante factor en een constante term na. We schrijven *Prim is* $O(n^4)$: de juiste definitie van O komt in de volgende sectie. Voor grote grafen is dat onnoemelijk veel beter dan voor het naief algoritme dat minimaal $O(n^{n-2})$ is. In feite is de complexiteit van de optimale implementatie van het algoritme van Prim beduidend beter dan $O(n^4)$.

Twee opmerkelijke feiten over Prim Het algoritme van Prim heeft twee interessante aspecten: de initialisatie bevat een toevals-component (we noemen het *niet-deterministisch*), en binnen de lus is er ook een niet-deterministische keuze als er meer dan één boog aan de voorwaarde voldoet. De berekende MOB is dus afhankelijk van dat toeval (een graaf kan immers meer dan één MOB hebben), maar de correctheid niet.

Dit algoritme is ook *gulzig*: er wordt telkens heel kortzichtig beslist welke boog nu toe te voegen (de kleinste) zonder het globale minimaliteitscriterium in het oog te houden. Het bewijs van de correctheid van het algoritme van Prim toont aan dat - dankzij de bewezen eigenschap - de gulzigheid ook leidt tot optimaliteit.

Gulzige algoritmen werken inderdaad niet altijd optimaal: denk aan een manier om een bepaald bedrag terug te geven met zo weinig mogelijk munten en stel dat je van elke muntsoort een onbeperkt aantal hebt. De gulzige methode bestaat erin om altijd als volgende munt de grootst mogelijke te nemen: je kijkt niet verder vooruit. Stel bijvoorbeeld dat het terug te geven bedrag gelijk is aan 8, en je hebt muntstukken met waarde 1,2,4 en 5, dan gebruik je met de gulzige methode 3 muntstukken: $5+2+1=8$. Maar het kan ook met slechts 2 munten, want $4+4=8$. Hier zijn twee kleine uitdagingen: aan welke voorwaarde moeten de waarden van de beschikbare munten voldoen zodat het gulzig algoritme altijd optimaal is? Voldoen onze Euromunten eraan?

Wat leren we hieruit? Een wiskundige stelling over grafen is essentieel om de correctheid van een algoritme aan te tonen. Het bewijs gebruikt logica en zijn typische afleidingsregels, en dikwijls ook inductie. Bovendien volgt ook uit de stelling dat het gulzige aspect van het algoritme correct is: dankzij de gulzigheid is het algoritme ook efficiënt. M.a.w., een wiskundig inzicht leidt tot een efficiënt algoritme. Dit is altijd zo: een efficiënt algoritme is altijd gebaseerd op een wiskundig inzicht.

Om af te sluiten Inleidingen tot grafentheorie zijn gemakkelijk te vinden op het net: gebruik *inleiding grafentheorie* of *introduction graph theory* in je zoekmachine. Die zoekmachine gebruikt overigens ook een pak wiskunde om snel relevante resultaten te geven. Grafentheorie wordt echt veel gebruikt in de algoritmiek, o.a. bij matching

problemen, netwerkstromingsproblemen, planningsproblemen ... en natuurlijk het kortste-pad probleem dat onze GPS oplost bij het plannen van een reis. De toepassingsgebieden van zulke graafgebaseerde algoritmen zijn ook heel divers: chemie, economie, sociale wetenschappen, productie ... Ons dagelijks leven steunt op algoritmen en die steunen op wiskunde.

Recurrentievergelijkingen en de efficiëntie van algoritmen

Hieronder hebben we op twee verschillende manieren één en dezelfde functie geïmplementeerd - we hebben een nieuwe programmaconstructie gebruikt: die wordt meteen duidelijk.

```
traag(n):                    |    rap(n):
  s = 0;                     |      s = n*(n+1)/2;
  for (i = 1 upto n) do      |      return s;
      s = s + i;             |
  return s;                  |
```

De constructie *for (x = a upto b) do* herhaalt een aantal keer wat achter de *do* staat: daarin heeft x bij het eerste doorlopen van de lus de waarde a, dan $a + 1$ ... tot de laatste keer wanneer $x == b$. Binnen een for-lus mag de i niet aangepast worden, dus weet je van tevoren hoe dikwijls de lus wordt uitgevoerd. Een for-lus kan je ook schrijven als een while-lus, maar dan heb je meer instructies nodig en is het minder vlug duidelijk wat er gebeurt: probeer het toch maar eens!
Het programma voor $traag(n)$ berekent de som van de getallen van 1 tot n, en we kennen daarvoor een gesloten formule: $rap(n)$ implementeert die.

In de veronderstelling dat een optelling even lang duurt als een vermenigvuldiging of een deling en een test - is dat echt zo? - zie je dat *traag(n)* $3*n+4$ operaties kost. Systematisch tellen levert inderdaad:

- 2 voor de initialisatie van s en i
- n keer 1 optellen bij i (kan je uitleggen waarom het wel degelijk n is en niet $(n-1)$?)
- $n+1$ keer testen of i niet groter gaat worden dan i
- n optellingen $s = s + i$
- één *return*

Je kan meer of minder gedetailleerd tellen, maar wat overblijft is dat *traag* een aantal operaties kost dat lineair is in n. We schrijven: $traag\ is\ O(n)$ - we spreken dat uit als *traag is oo n*. De exacte definitie van O geven we later.
Voor $rap(n)$ zie je al snel dat die 4 operaties kost, onafhankelijk van n, en dat noteren we met $O(1)$.
Het is intuïtief al duidelijk dat als n groot genoeg is, $traag(n)$ meer operaties nodig heeft dan $rap(n)$. Die intuïtie wordt ondersteund door de definitie van $f(n)\ is\ O(g(n))$ (wat we lezen als f *is oo* g), waarbij f en g positieve functies zijn op $\mathbb{N}$), namelijk:
$\exists c > 0, n_0 > 0, \forall n > n_0 : f(n) \leq cg(n)$. We zeggen ook: op een constante factor na wordt f asymptotisch gedomineerd door g. Informeel betekent het dat f niet rapper groeit dan g. Volgens de definitie is $999 * n + 1000\ O(n^2)$. Je kan in de definitie bijvoorbeeld n_0 gelijk nemen aan 1000, en $c = 1$. Maar ook $n_0 = 999$ en $c = 1001$ laten zien dat $999 * n + 1000\ is\ O(n^2)$.
Bepaal nu zelf welke van de volgende uitspraken waar zijn:

- $nlog(n)\ is\ O(n^2)$

- indien f *is* $O(n^{17})$ dan ook f *is* $O(n^{18})$
- $n^8 =$ niet $O(n^7)$
- $1000 * n^{23} + 666 * n^{22}$ *is* $O(0,5 * n^{23})$

Het asymptotisch gedrag van het aantal elementaire operaties dat een algoritme uitvoert, geeft een idee over de efficiëntie van het algoritme voor grotere initiële gegevens. Dat vinden we belangrijk, want als we met een programma een optimale treinregeling kunnen maken voor 10 stations in 1 minuut, dan willen we kunnen inschatten of het voor 20 stations 2 minuten zal duren, of misschien veel langer.
De $O(g)$ van een algoritme zegt dus iets over de efficiëntie van een algoritme, of in ons vakjargon, iets over de complexiteit(sklasse) van het algoritme. Hoe bepalen we de complexiteit van een algoritme? Hieronder geven we een methode die werkt voor heel veel algoritmen: recurrentievergelijkingen, soms ook recursievergelijkingen genoemd. Als voorbeeld een stukje code dat voor een gegeven waarde van n een waarde berekent:

```
f(n):
   s = 0;
   for (i = 1 upto n) do
      for (j = i upto n) do
            <doe iets elementair met s>
   return s;
```

Wat in de binnenste lus juist gebeurt met s hebben we niet neergeschreven, omdat we nu enkel geinteresseerd zijn in hoe dikwijls het algoritme de elementaire operatie uitvoert, want dat geeft aan in welke klasse het algoritme zit.
Noteer met $A(n)$ het aantal elementaire operaties dat het algoritme uitvoert voor input n. Dan weten we dat $A(0) = 4$ (de initialisatie van s, de initialisatie op 1 van i, de test of $i > n$, en de return). We stellen een vergelijking op die $A(n+1)$ bepaalt in termen van $A(n)$ (en misschien $A(n-1)$ ...). We concentreren ons even op de dubbele lus:

```
for (i = 1 upto (n+1)) do
   for (j = i upto (n+1)) do
         <doe iets elementair met s>
```

Het is misschien al op het zicht duidelijk dat die elementaire actie een kwadratisch aantal keer in n wordt uitgevoerd, maar hier willen we een algemene oplossingsmethode laten zien.
Je kan inzien dat bovenstaand programma hetzelfde doet als

```
for (i = 1 upto n) do
   for (j = i upto n) do
         <doe iets elementair met s>

for (i = 1 upto n) do
   j = (n+1);
   <doe iets elementair met s>

i = (n+1);
for (j = (n+1) upto (n+1)) do
    <doe iets elementair met s>
```

Je merkte misschien dat de volgorde waarin de elementaire operaties uitgevoerd worden veranderde, maar omdat we enkel willen tellen hoeveel er uitgevoerd worden, maakt dat niet uit.
De eerste dubbele lus geeft ons $A(n)$, de twee andere lussen kosten $2 * n$ en 6 (waarbij de exacte waarden van die constanten 2 en 6 niet zo belangrijk zijn). Dus weten we dat $A(n+1) = A(n) + 2 * n + 6$, met beginvoorwaarde $A(0) = 4$. Deze recurrentievergelijking gelijkt wat op een differentiaalvergelijking. Het is een lineaire, niet-homogene recurrentievergelijking van de eerste orde. Het niet-homogene stuk is een veelterm van de eerste graad $(2 * n + 6)$ en de algemene oplossingsmethode voor zulke vergelijkingen geeft als resultaat dat $A(n) = \alpha *$

$n^2+\beta*n+\gamma$, waarbij α, β, γ bepaald worden uit het niet-homogene deel en de beginvoorwaarde. De exacte oplossing is niet zo belangrijk voor de complexiteitsbepaling: $A(n)$ *is* $O(n^2)$.
Recurrentievergelijkingen worden geclassificeerd aan de hand van hun graad en lineariteit, en een stukje theorie i.v.m. recurrentievergelijkingen steunt op lineaire algebra (vectorruimten, basis, onafhankelijkheid van vectoren, aantal oplossingen ...), de ontbinding van polynomen, hun nulpunten (in het complexe vlak) en goniometrische functies.
Recurrentievergelijkingen kunnen dikwijls op systematische wijze van een programma of algoritme worden afgeleid, maar zijn niet altijd even gemakkelijk op te lossen. Zo is de recurrentievergelijking van binair zoeken (zie elders in deze Loep) $A(n) = A(n/2) + 1$: de oplossing daarvan is $O(log_2(n))$.

Wat wilden we laten zien? Het asymptotisch gedrag van een algoritme is van belang voor informatici. Het kan dikwijls bepaald worden door een recurrentievergelijking op te stellen, en die op te lossen voor de gegeven beginvoorwaarden. De theorie achter zulke vergelijkingen is wiskundig. Het hele domein van complexiteitstheorie is overigens zeer wiskundig en van belang in cryptografie en het algoritmisch beveiligen van informatie en transacties op het internet. Bovendien ligt één van de *million dollar problems* van het *Clay Mathematics Institute* in de complexiteitstheorie, namelijk de vraag of P gelijk is aan NP.

Eindigheid van een algoritme

We belichten hier eerder oppervlakkig enkele onderwerpen die relevant zijn bij de studie van het *stoppen* van een procedure. We lieten voorheen al zien dat dit wiskundig bekeken kan worden, los van de correctheid van de procedure. Als je een recurrentievergelijking kan opstellen (en oplossen) voor het aantal stappen dat een algoritme uitvoert, dan heb je daarmee natuurlijk ook het bewijs dat het programma stopt voor elke input. Soms kan je bewijzen dat een programma niet stopt voor sommige input. De functie

```
f(n):
   if (n > 0) then return f(n+1);
```

is een triviaal voorbeeld van zulk een functie. Maar pas op: in een computer wordt dikwijls gerekend modulo 2^{32} of 2^{64}, en dan stop die f voor elke input!
De theorie van formele talen leert ons bovendien:

- er bestaat geen eindige bewijsmethode die van elk programma correct zijn terminatiegedrag kan bepalen: dit is het antwoord op het fameuze stopprobleem (of het *Haltingprobleem*) dat A. Turing in de jaren 1930 bestudeerde; de essentie van de moeilijkheid zit in de *while* herhalingsopdracht en zowat alle programmeertalen hebben die; elke eindigheidsbewijsmethode is dus inherent onvolledig; zulke methoden zijn bijna altijd gebaseerd op het bestaan van een goede orde op de grootheden in het programma met daarbij een bewijs dat die *kleiner* worden. Dat ligt niet altijd voor de hand. Zo stopt het programma $Ack(n, m)$ dat de Ackermann-functie berekent voor elke positieve gehele input, maar het is een uitdaging dat te bewijzen.

```
Ack(m,n):
  if (m == 0) then return n+1;
  else if (n == 0) then return A(m-1,1);
  else return A(m-1,A(m,n-1));
```

Die Ack-functie heeft overigens een belangrijke rol gespeeld in het begrijpen en definiëren van de notie *berekenbare functie*, in de eerste plaats door de wiskundige K. Gödel en door A. Turing, maar met wortels in wat wiskundigen het *programma van D. Hilbert* noemen

- van een deelklasse van de programma's is het evident dat die op elke input stoppen: programma's met enkel de *if-then-else* en de *for lus* (en zonder recursieve functie-oproepen); met zulke programma's kan je al heel veel doen, maar deze klasse is niet Turing-compleet: er bestaan functies die je er niet mee kan programmeren; de Ackermann-functie is daarvan een voorbeeld

Misschien denk je nu: zelfs al kunnen we niet *algoritmisch* bepalen van **elk** programma of het stopt, we kunnen dat van een specifiek programma altijd wel aantonen, want voor een specifiek programma kunnen we slimme ad-hoc stellingen bewijzen. En toch, hieronder een bedrieglijk eenvoudig programma waarvan het eindigheidsbewijs nog steeds ontsnapt aan informatici en wiskundigen:

```
collatz(n):
  while (n <> 1) do
      if (even(n)) then n := n / 2; % gehele deling
                   else n := 3*n + 1;
  return 1;
```

Voor een gegeven waarde van n doorloopt n tijdens de uitvoering van het programma een Collatz-rij. Bijvoorbeeld voor gegeven input 19, krijg je $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$. Je kan het programma gemakkelijk voor veel waarden van n uitproberen en je zal zien dat het programma elke keer stopt, maar bewijzen dat het voor elke input stopt is een ander paar mouwen. Zoals P. Erdös zegde: *Mathematics is not yet ready for such problems.*

Informatica toepassingen in de wiskunde?

Vermits wiskunde in de eerste plaats gaat over het bewijzen van nieuwe stellingen (denk ik toch) kan je je afvragen hoe je daarbij informatica kan gebruiken? Het meest bekende voorbeeld is waarschijnlijk het 4-kleuren probleem: kan elke vlakke kaart gekleurd worden met 4 kleuren? Voor heel wat wiskundigen is dat *computerbewijs* niet voldoende en ze hebben evenveel afkeer van geprogrammeerde algemene automatische stellingbewijzers. Ik begrijp dat standpunt ergens wel, maar ik treed het niet bij. Er zijn echter ook minder controversiële manieren om informatica te gebruiken in de wiskunde, en van één ervan heb ik zelf gebruik gemaakt om een aantal wiskundige stellingen te *ontdekken*, met dan daarna wel een handmatig bewijs ervan. Dat ging als volgt: door exhaustief zoeken ontdekten we een bepaald patroon in de regels van Sudoku. Dat patroon werd dan verheven tot hypothese, of conjectuur, en later wiskundig correct bewezen: het ging over de overbodigheid van sommige Sudoku-regels.

Bronnen

`nl.wikipedia.org/wiki/Algoritme_van_Euclides` bekeken op 9 oktober 2013

Aigner, Martin; Ziegler, Günter (2009). Proofs from THE BOOK (4th ed.). Berlin, New York: Springer-Verlag, of www.iecn.u-nancy.fr/~chassain/djvu/Proofs-from-the-Book-2004.pdf met het resultaat van Cayley

en.wikipedia.org/wiki/Boruvka's_algorithm bekeken op 5 augustus 2013

nl.wikipedia.org/wiki/Algoritme_van_Prim bekeken op 5 augustus 2013

Demoen, B. en Garcia de la Banda, M. *Less constraints is still Sudoku*, Journal of Theory and Practice of Logic Programming, to appear 2013, lirias.kuleuven.be/handle/123456789/353637

claymath.org/millennium/P_vs_NP

en.wikipedia.org/wiki/Prim%27s_algorithm#Time_complexity

nl.wikipedia.org/wiki/Vierkleurenstelling